Computing Abstract Distances in Logic Programs*

Ignacio Casso[1,2][0000−0001−9196−7951], José F. Morales[1][0000−0001−6098−3895], Pedro López-García[1,3][0000−0002−1092−2071], Roberto Giacobazzi[1,4][0000−0002−9582−3960], and Manuel V. Hermenegildo[1,2][0000−0002−7583−323X]

[1] IMDEA Software Institute
[2] T. University of Madrid (UPM)
[3] Spanish Council for Scientific Research (CSIC)
[4] University of Verona, Italy

Abstract. Abstract interpretation is a well-established technique for performing static analyses of logic programs. However, choosing the abstract domain, widening, fixpoint, etc. that provides the best precision-cost trade-off remains an open problem. This is in a good part because of the challenges involved in measuring and comparing the precision of different analyses. We propose a new approach for measuring such precision, based on defining distances in abstract domains and extending them to distances between whole analyses of a given program, thus allowing comparing precision across different analyses. We survey and extend existing proposals for distances and metrics in lattices or abstract domains, and we propose metrics for some common domains used in logic program analysis, as well as extensions of those metrics to the space of whole program analyses. We implement those metrics within the CiaoPP framework and apply them to measure the precision of different analyses on both benchmarks and a realistic program.

Keywords: Abstract interpretation, static analysis, logic programming, metrics, distances, complete lattices, program semantics.

1 Introduction

Many practical static analyzers for (Constraint) Logic Programming ((C)LP) are based on the theory of Abstract Interpretation [8]. The basic idea behind this technique is to interpret (i.e., execute) the program over a special abstract domain to obtain some abstract semantics of the program, which will over-approximate every possible execution in the standard (concrete) domain. This

* Research partially funded by MINECO TIN2015-67522-C3-1-R *TRACES* project, and the Madrid P2018/TCS-4339 *BLOQUES-CM* program. We are also grateful to the anonymous reviewers for their useful comments.

makes it possible to reason safely (but perhaps imprecisely) about the properties that hold for all such executions. As mentioned before, abstract interpretation has proved practical and effective for building static analysis tools, and in particular in the context of (C)LP [32,21,40,5,12,4,30,16,26]. Recently, these techniques have also been applied successfully to the analysis and verification of other programming paradigms by using (C)LP (Horn Clauses) as the intermediate representation for different compilation levels, ranging from source to bytecode or ISA [31,2,35,17,27,10,19,3,29,25].

When designing or choosing an abstract interpretation-based analysis, a crucial issue is the trade-off between cost and precision, and thus research in new abstract domains, widenings, fixpoints, etc., often requires studying this trade-off. However, while measuring analysis cost is typically relatively straightforward, having effective precision measures is much more involved. There have been a few proposals for this purpose, including, e.g., probabilistic abstract interpretation [13] and some measures in numeric domains [28,39], [5] but they have limitations and in practice most studies come up with ad-hoc measures for measuring precision. Furthermore, there have been no proposals for such measures in (C)LP domains.

We propose a new approach for measuring the precision of abstract interpretation-based analyses in (C)LP, based on defining *distances in abstract domains* and extending them to *distances between whole analyses of a given program*, which allow comparison of precision across different analyses. Our contributions can be summarized as follows: We survey and extend existing proposals for distances in lattices and abstract domains (Sec. 3). We then build on this theory and ideas to propose distances for common domains used in (C)LP analysis (Sec. 3.2). We also propose a principled methodology for comparing quantitatively the precision of different abstract interpretation-based analyses of a whole program (Sec. 4). This methodology is parametric on the distance in the underlying abstract domain and only relies in a unified representation of those analysis results as AND-OR trees. Thus, it can be used to measure the precision of new fixpoints, widenings, etc. within a given abstract interpretation framework, not requiring knowledge of its implementation. Finally, we also provide experimental evidence about the appropriateness of the proposed distances (Sec. 5).

2 Background and Notation

Lattices: A *partial order* on a set X is a binary relation $\sqsubseteq$ that is reflexive, transitive, and antisymmetric. The *greatest lower bound* or *meet* of a and b, denoted by $a \sqcap b$, is the greatest element in X that is still lower than both of them ($a \sqcap b \sqsubseteq a$, $a \sqcap b \sqsubseteq b$, $(c \sqsubseteq a \ \wedge c \sqsubseteq b \implies c \sqsubseteq a \sqcap b)$). If it exists, it is unique. The *least upper bound* or *join* of a and b, denoted by $a \sqcup b$, is the smallest element in X that is still greater than both of them ($a \sqsubseteq a \sqcup b$, $b \sqsubseteq a \sqcup b$,

[5] Some of these attempts (and others) are further explained in the related work section (Section 6).

$(a \sqsubseteq c \wedge b \sqsubseteq c \implies a \sqcup b \sqsubseteq c)$). If it exists, it is unique. A partially ordered set (poset) is a couple $(X, \sqsubseteq)$ such that the first element X is a set and the second one is a partial order relation on X. A *lattice* is a poset for which any two elements have a meet and a join. A lattice L is complete if, extending in the natural way the definition of supremum and infimum to subsets of L, every subset S of L has both a supremum $sup(S)$ and an infimum $inf(S)$. The maximum element of a complete lattice, $\sup(L)$ is called *top* or $\top$, and the minimum, $\inf(L)$ is called *bottom* or $\bot$.

Galois Connections: Let $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ be two posets. Let $f : L_1 \longrightarrow L_2$ and $g : L_2 \longrightarrow L_1$ be two applications such that:

$$\forall x \in L_1, y \in L_2 : f(x) \sqsubseteq_2 y \iff x \sqsubseteq_1 g(y)$$

Then the quadruple $\langle L_1, f, L_2, g\rangle$ is a *Galois connection*, written $L_1 \underset{f}{\overset{g}{\leftrightarrows}} L_2$. If $f \circ g$ is the identity, then the quadruple is called a *Galois insertion.*

Abstract Interpretation and Abstract Domains: Abstract interpretation [8] is a well-known static analysis technique that allows computing sound over-approximations of the semantics of programs. The semantics of a program can be described in terms of the *concrete domain*, whose values in the case of (C)LP are typically sets of variable substitutions that may occur at runtime. The idea behind abstract interpretation is to interpret the program over a special abstract domain, whose values, called *abstract substitutions*, are finite representations of possibly infinite sets of actual substitutions in the concrete domain. We will denote the concrete domain as D, and the abstract domain as D_α. We will denote the functions that relate sets of concrete substitutions with abstract substitutions as the *abstraction* function $\alpha : D \longrightarrow D_\alpha$ and the *concretization* function $\gamma : D_\alpha \longrightarrow D$. The concrete domain is a complete lattice under the set inclusion order, and that order induces an ordering relation in the abstract domain herein represented by "$\sqsubseteq$." Under this relation the abstract domain is usually a complete lattice and $(D, \alpha, D_\alpha, \gamma)$ is a Galois insertion. The abstract domain is of finite height or alternatively it is equipped with a *widening operator*, which allows for skipping over infinite ascending chains during analysis to a greater fixpoint, achieving convergence in exchange for precision.

Metric: A metric on a set S is a function $d : S \times S \to \mathbb{R}$ satisfying:

- Non-negativity: $\forall x, y \in S,\ d(x, y) \geq 0.$
- Identity of indiscernibles: $\forall x, y \in S,\ d(x, y) = 0 \iff x = y.$
- Symmetry: $\forall x, y \in S,\ d(x, y) = d(y, x).$
- Triangle inequality: $\forall x, y, z \in S,\ d(x, z) \leq d(x, y) + d(y, z).$

A set S in which a metric is defined is called a metric space. A pseudometric is a metric where two elements which are different are allowed to have distance 0. We call the left implication of the identity of indiscernibles, weak identity of indiscernibles. A well-known method to extend a metric $d : S \times S \longrightarrow \mathbb{R}$ to a distance in 2^S is using the Hausdorff distance, defined as:

$$\mathrm{d}_H(A, B) = \max\left\{\sup_{a \in A} \inf_{b \in B} \mathrm{d}(a, b), \sup_{b \in B} \inf_{a \in A} \mathrm{d}(a, b)\right\}$$

3 Distances in Abstract Domains

As anticipated in the introduction, our distances between abstract interpretation-based analyses of a program will be parameterized by a distance in the underlying abstract domain, which we assume to be a complete lattice. In this section we propose a few such distances for relevant logic programming abstract domains. But first we review and extend some of the concepts that arise when working with lattices or abstract domains as metric spaces.

3.1 Distances in Lattices and Abstract Domains

When defining a distance in a partially ordered set, it is necessary to consider the compatibility between the metric and the structure of the lattice. This relationship will suggest new properties that a metric in a lattice should satisfy. For example, a distance in a lattice should be *order-preserving*, that is, $\forall a,b,c \in D \textit{ with } a \sqsubseteq b \sqsubseteq c, \textit{ then } d(a,b), d(b,c) \leq d(a,c)$. It is also reasonable to expect that it fulfills what we have called the diamond inequality, that is, $\forall a,b,c,d \in D \textit{ with } c \sqcap d \sqsubset a \sqcap b,\ a \sqcup b \sqsubset c \sqcup d, \textit{ then } d(a,b) \leq d(c,d)$. But more importantly, this relationship will suggest insights for constructing such metrics.

One such insight is precisely defining a partial metric $d_{\sqsubseteq}$ only between elements which are related in the lattice, which is arguably easier, and to extend it later to a distance between arbitrary elements x, y, as a function of $d_{\sqsubseteq}(x, x \sqcap y)$, $d_{\sqsubseteq}(y, x \sqcap y)$, $d_{\sqsubseteq}(x, x \sqcup y)$, $d_{\sqsubseteq}(x, x \sqcup y)$ and $d_{\sqsubseteq}(x \sqcap y, x \sqcup y)$. Jan Ramon et al. [38] show under which circumstances $d_{\sqsubseteq}(x, x \sqcup y) + d_{\sqsubseteq}(y, x \sqcup y)$ is a distance, that is, when $d_{\sqsubseteq}$ is order-preserving and fulfills $d_{\sqsubseteq}(x, x \sqcup y) + d_{\sqsubseteq}(y, x \sqcup y) \leq d_{\sqsubseteq}(x, x \sqcap y) + d_{\sqsubseteq}(y, x \sqcap y)$.

In particular, one could define a monotonic size $\mathit{size} : L \to \mathbb{R}$ in the lattice and define $d_{\sqsubseteq}(a,b)$ as $\mathit{size}(b) - \mathit{size}(a)$. Gratzer [18] shows that if the size fulfills $\mathit{size}(x) + \mathit{size}(y) = \mathit{size}(x \sqcap y) + \mathit{size}(x \sqcup y)$, then $d(x,y) = \mathit{size}(x \sqcup y) - \mathit{size}(x \sqcap y)$ is a metric. De Raedt [11] shows that $d(x,y) = \mathit{size}(x) + \mathit{size}(y) - 2 \cdot \mathit{size}(x \sqcup y)$ is a metric iff $\mathit{size}(x) + \mathit{size}(y) \leq \mathit{size}(x \sqcap y) + \mathit{size}(x \sqcup y)$, and an analogous result with $d(x,y) = \mathit{size}(x) + \mathit{size}(y) - 2 \cdot \mathit{size}(x \sqcup y)$ and $\geq$ instead of $\leq$. Note that the first distance is the equivalent of the *symmetric difference distance* in finite sets, with $\sqsubseteq$ instead of $\subseteq$ and *size* instead of the cardinal of a set. Similar distances for finite sets, such as the Jaccard distance, can be translated to lattices in the same way. Another approach to defining $d_{\sqsubseteq}$ that follows from the idea of using the lattice structure, is counting the steps between two elements (i.e., the number of edges between both elements in the Hasse diagram of the lattice). This was used by Logozzo [28].

When defining a distance not just in any lattice, but in an actual abstract domain (*abstract distance* from now on), it is also necessary to consider the relation of the abstract domain with the concrete domain (i.e., the Galois connection), and how an abstract distance is interpreted under that relation. In that sense, we can observe that a distance $d_{D_\alpha} : D_\alpha \times D_\alpha \to \mathbb{R}^+$ in an abstract domain will induce a distance $d_D^\alpha : D \times D \to \mathbb{R}^+$ in the concrete one, as $d_D^\alpha(A,B) = d_{D_\alpha}(\alpha(A), \alpha(B))$, and the other way around: a distance $d_D : D \times D \to \mathbb{R}^+$ in the concrete domain induces an abstract distance $d^{\gamma}_{D_\alpha} : D_\alpha \times D_\alpha \to \mathbb{R}^+$ in the abstract one, as $d^{\gamma}_{D_\alpha}(a,b) = d_D(\gamma(a), \gamma(b))$. Thus, an abstract distance can be interpreted as an abstraction of a distance in the concrete domain, or as a way to define a distance in it, and it is clear that it is when interpreted that way that an abstract distance makes most sense from a program semantics point of view.

It is straightforward to see (and we show in [6]) that these induced distances inherit most metric and order-related properties. In particular, if a distance d_D in the concrete domain is a metric, its abstraction d_{D_α} is a pseudo-metric in the abstract domain, and a full metric if the Galois connection between D and D_α is a Galois insertion. This allows us to define distances d_α in the abstract domain from distances d the concrete domain, as $d_\alpha(a,b) = d(\gamma(a), \gamma(b))$. This approach might seem of little applicability, due to the fact that concretizations will most likely be infinite and we still need metrics in the concrete domain. But in the case of logic programs, such metrics for Herbrand terms already exist (e.g., [23,36,38]), and in fact we show later a distance for the *regular types* domain that can be interpreted as an extension of this kind, of the distance proposed by Nienhuys-Cheng [36] for sets of terms.

Finally, recall that a metric in the Cartesian product of lattices can be easily derived from existing distances in each lattice, for example as the 2-norm or any other norm of the vector of distances component to component. This is relevant because many abstract domains, such as those that are combinations of two different abstract domains, or non-relational domains which provide an abstract value from a lattice for each variable in the substitution, are of such form (modulo equivalent abstract values, i.e., those with the same concretization). It is straightforward to see that in this case those classical extensions of distances to the Cartesian product will still be metrics and will also inherit lattice-related properties such as being *order-preserving.*

3.2 Distances in Logic Programming Domains

We now propose some distances for two well-known abstract domains used in (C)LP, following the considerations presented in the previous section.

Sharing Domain: The `sharing` domain [24,32] is a well-known domain for analyzing the sharing (aliasing) relationships between variables and grounding in logic programs. It is defined as $2^{2^{Pvar}}$, that is, an abstract substitution for a clause is defined to be *a set of sets of program variables* in that clause, where each set indicates that the terms to which those variables are instantiated at run-time might share a free variable. More formally, we define $Occ(\theta, U) = \{X | X \in dom(\theta), U \in vars(X\theta)\}$, the set of all program variables $X \in Pvar$ in the clause such that the variable $U \in Uvar$ appears in $X\theta$. We define the abstraction of a substitution θ as $\mathcal{A}_{sharing}(\theta) = \{Occ(\theta, U) \mid U \in Uvar\}$, and extend it to sets of substitutions. The order induced by this abstraction in $2^{2^{Pvar}}$ is the set inclusion, the join, the set union, and the meet, the set intersection. As an example, $\top = 2^{Pvar}$, a program variable that does not appear in any set is guaranteed to be ground, or two variables that never appear in the same set are guaranteed to not share. The complete definition can be found in [24,32]).

Following the approach of the previous section, we define this monotone size in the domain: $size(a) = |a| + 1, size(\bot) = 0$. It is straightforward to check that $\forall a, b \in 2^{2^{Pvar}},\ size(a) + size(b) = size(a \sqcap b) + size(a \sqcup b)$. Therefore the following distance is a metric and order-preserving: $d_{share}(Sh_1, Sh_2) =$

$$size(Sh_1 \cup Sh_2) - size(Sh_1 \cap Sh_2) = |Sh_1 \cup Sh_2| - |Sh_1 \cap Sh_2|$$

We would like our distance to be in a normalized range $[0, 1]$, and for that we divide it between $d(\bot, \top) = 2^n + 1$, where $n = |V|$ denotes the number of variables in the domain of the substitutions. This yields the following final distance, which is a metric by construction:

$$d_{share}(Sh_1, Sh_2) = (|(Sh_1 \cup Sh_2)| - |size(Sh_1 \cap Sh_2)|)/(2^n + 1)$$

Regular-Type Domain: Another well-known domain for logic programs is the *regular types* domain [9], which abstracts the shape or type of the terms to which variables are assigned at run time. It associates each variable with a deterministic context free grammar that describes its shape, with the possible functors and atoms of the program as terminal symbols. A more formal definition can be found in [9]. We will write abstract substitutions as tuples $\langle G_1, \ldots, G_n \rangle$, where $G_i = (\mathcal{T}_i, \mathcal{F}_i, \mathcal{R}_i, S_i)$ is the grammar that describes the term associated to the i-th variable in the substitution. We propose to use as a basis the Hausdorff distance in the concrete domain, using the distance between terms proposed in [36], i.e.,

$$d_{term}(f(x_1, \ldots, x_n), g(y_1, \ldots, y_m)) = \begin{cases} 1 \quad if \quad f/n \neq g/m \\ else: \quad p \sum_{i=1}^{n} \frac{1}{n} d_{term}(x_i, y_i) \end{cases}$$

where p is a parameter of the distance. As the derived abstract version, we propose the following recursive distance between two types or grammars G_a, G_b, where $G|_T$ is the grammar G with initial symbol T instead of S:

$$d'(G_a, G_b) = \begin{cases} 1\ if\ \exists\ (S_a \to f(T_1, \ldots, T_n)) \in \mathcal{R}_a \wedge \nexists (S_b \to f(T'_1, \ldots, T'_n)) \in \mathcal{R}_b \\ 1\ if\ \exists\ (S_b \to f(T_1, \ldots, T_n)) \in \mathcal{R}_b \wedge \nexists (S_a \to f(T'_1, \ldots, T'_n)) \in \mathcal{R}_a \\ else: max\{p \sum_{i=1}^{n} \frac{1}{n} d'(G_a|_{T_i}, G_b|_{T'_i}) \mid (S_a \to f(T_1, \ldots, T_n)) \in \mathcal{R}_a, \\ \qquad (S_b \to f(T'_1, \ldots, T'_n)) \in \mathcal{R}_b\} \end{cases}$$

We also extend this distance between types to distance between substitutions in the abstract domain as follows: $d(\langle T_1, \ldots, T_n \rangle, \langle T'_1, \ldots, T'_n \rangle) = \sqrt{d'(T_1, T'_1)^2 + \ldots + d'(T_n, T'_n)^2}$. Since d' is the abstraction of the Hausdorff distance with d_{term}, which it is proved to be a metric in [36], d' is a metric too. [6] Therefore d is also a metric, since it is its extension to the cartesian product.

4 Distances between Analyses

We now attempt to extend a distance in an abstract domain to distances between results of different abstract interpretation-based analyses of the same program

[6] Actually that only guarantees that d' is a pseudo-metric. Proving that it is indeed a metric is more involved and not really relevant to our discussion.

over that domain. In the following we will assume (following most "top-down" analyzers for (C)LP programs [32,4,16,26]) that the result of an analysis for a given entry (i.e., an initial predicate P, and an initial call pattern or abstract query λ_c), is an AND-OR tree, with root the OR-node $\langle P, \lambda_c, \lambda_s \rangle_\vee$, where λ_s is the abstract substitution computed by the analysis for that predicate given that initial call pattern. An AND-OR tree alternates AND-nodes, which correspond to clauses in the program, and OR-nodes, which correspond to literals in those clauses. An AND-node is a triplet $\langle C, \beta_{entry}, \beta_{exit} \rangle_\wedge$, with C a clause $Head : -L_1, ..., L_n$ and with $\beta_{entry}, \beta_{exit}$ the abstract entry and exit substitutions for that clause. It has an OR-node $\langle L_i, \lambda_c^i, \lambda_s^i \rangle_\vee$ as child for each literal L_i in the clause, where $\lambda_c^1 = \beta_{entry}$, $\lambda_c^{i+1} = \lambda_s^i$ for $i = 2 \ldots n$, and $\beta_{exit} = \lambda_s^n$. An OR-node is a triplet $\langle L, \lambda_c, \lambda_s \rangle_\vee$, with L a literal in the program, which is a call to a predicate P, and λ_c, λ_s the abstract call and success substitutions for that goal. It has one AND-node $\langle C_j, \beta_{entry}^j, \beta_{exit}^j \rangle_\wedge$ as child for each clause C_j in the definition of P, where β_{entry}^j is derived from λ_c by projecting and renaming to the variables in the clause C_j, and λ_s is obtained from $\{\beta_{exit}^j\}$ by extending and renaming each exit substitution to the variables in the calling literal L and computing the least upper bound of the results. This tree is the abstract counterpart of the resolution trees that represent concrete top-down executions, and represents a possibly infinite set of those resolution trees at once. The tree will most likely be infinite, but can be represented as a finite cyclic tree. We denote the children of a node T as $ch(T)$ and its triplet as $val(T)$.

Example 1. Let us consider as an example the simple quick-sort program (using difference lists) in Fig. 1, which uses an *entry* assertion to specify the initial abstract query of the analysis [37]. If we analyze it with a simple *groundness* domain (with just two values `g` and `ng`, plus $\top$ and $\bot$), the result can be represented with the cyclic tree shown in Fig. 1. That tree is a finite representation of an infinite abstract AND-OR tree. The nodes in layers 1,3 and 5 represent OR-nodes, and the ones in layers 2,4 and 6, AND-nodes, where `p/i/j` corresponds to the j-th clause of predicate `p/i`. The actual values of the nodes are specified below the tree. $\square$

We propose three distances between AND-OR trees S_1, S_2 for the same entry, in increasing order of complexity, and parameterized by a distance d_α in the underlying abstract domain. We also discuss which metric properties are inherited by these distances from d_α. Note that a good distance for measuring precision should fulfill the identity of indiscernibles.

Top Distance. The first consists in considering only the roots of the top trees, $\langle P, \lambda_c, \lambda_s^1 \rangle_\vee$ and $\langle P, \lambda_c, \lambda_s^2 \rangle_\vee$, and defining our new distance as $d(S_1, S_2) = d_\alpha(\lambda_s^1, \lambda_s^2)$. This distance ignores too much information (e.g., if the entry point is a predicate `main/0`, the distance would only distinguish analyses that detect failure from analysis which do not), so it is not appropriate for measuring analysis precision, but it is still interesting as a baseline. It is straightforward to see that it is a pseudometric if d_α is, but will not fulfill the identity of indiscernibles even if d_α does.

```
:- module(quicksort,
        [quicksort/2],
        [assertions]).
:- use_module(partition,
        [partition/4]).

:- entry quicksort(Xs,Ys)
   : (ground(Xs), var(Ys)).

quicksort(Xs,Ys) :-
     qs(Xs,Ys,[]).

qs([],Ys,Ys).
qs([X|Xs],Ys,Tail) :-
     partition(Xs,X,L,R),
     qs(R,R2,Tail),
     qs(L,Ys,[X|R2]).
```

(1) $\langle quicksort(Xs,Ys),\ \ \{Xs/g,Ys/ng\},\ \ \{Xs/g,Ys/g\}\rangle_{\vee}$

(2) $\langle quicksort/2/1,\ \ \{Xs/g,Ys/ng\},\ \ \{Xs/g,Ys/g\}\rangle_{\wedge}$

(3) $\langle qs(Xs,Ys,[]),\ \ \{Xs/g,Ys/ng\},\ \ \{Xs/g,Ys/g\}\rangle_{\vee}$

(4) $\langle qs/3/1,\ \ \{Ys/g\},\ \ \{Ys/g\}\rangle_{\wedge}$

(5) $\langle qs/3/2,\ \ \{X/g,Xs/g,Ys/ng,Tail/g,L/ng,R/ng,R2/ng\},$
$\{X/g,Xs/g,Ys/g,Tail/g,L/g,R/g,R2/g\}\rangle_{\wedge}$

(6) $\langle partition(Xs,X,L,R),\ \ \{Xs/g,X/g,L/ng,R/ng\},\ \ \{Xs/g,X/g,L/g,R/g\}\rangle_{\vee}$

(7) $\langle qs(R,R2,Tail),\ \ \{R/g,R2/ng,Tail/g\},\ \ \{R/g,R2/g,Tail/g\}\rangle_{\vee}$

(8) $\langle qs(L,Ys,[X|R2]),\ \ \{L/g,Ys/ng,X/g,R2/g\},\ \ \{L/g,Ys/g,X/g,R2/g\}\rangle_{\vee}$

(9) $\langle qs/3/1,\ \ \{Ys/g\},\ \ \{Ys/g\}\rangle_{\wedge}$

(10) $\langle qs/3/2,\ \ \{X/g,Xs/g,Ys/ng,Tail/ng,L/ng,R/ng,R2/ng\},$
$\{X/g,Xs/g,Ys/g,Tail/gL/g,R/g,R2/g\}\rangle_{\wedge}$

(11) $\langle qs/3/1,\ \ \{Ys/g\},\ \ \{Ys/g\}\rangle_{\wedge}$

(12) $\langle qs/3/2,\ \ \{X/g,Xs/g,Ys/ng,Tail/ng,L/ng,R/ng,R2/ng\},$
$\{X/g,Xs/g,Ys/g,Tail/gL/g,R/g,R2/g\}\rangle_{\wedge}$

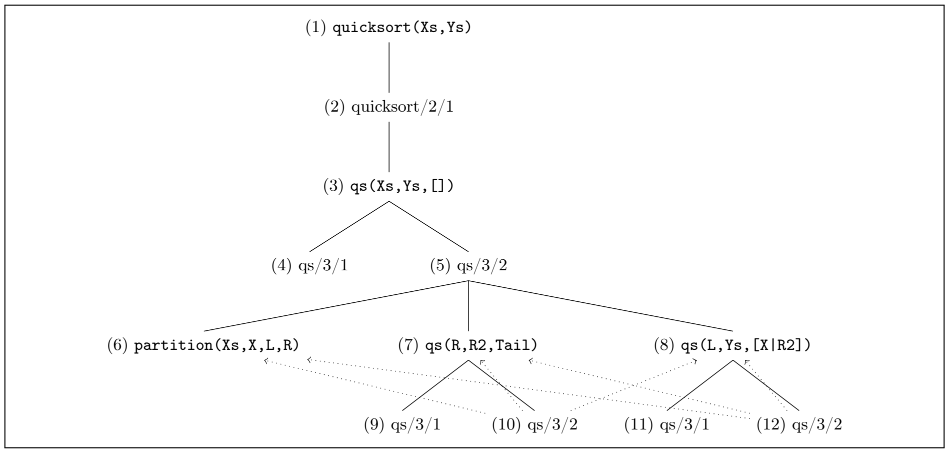

Fig. 1. Analysis of `quicksort/2` (using difference lists).

Flat Distance. The second distance considers all the information inferred by the analysis for each program point, but forgetting about its context in the AND-OR tree. In fact, analysis information is often used this way, i.e., considering only the substitutions with which a given literal in the program can be called or succeeds, and not which traces lead to those calls (path insensitivity). We define a distance between a program point or literal PP in two analysis S_1, S_2

$$d_{PP}(S_1,S_2) = \frac{1}{2}(d_\alpha(\bigsqcup_{\lambda\in PP_c^1}\lambda,\ \bigsqcup_{\lambda\in PP_c^2}\lambda) + d_\alpha(\bigsqcup_{\lambda\in PP_s^1}\lambda,\ \bigsqcup_{\lambda\in PP_s^2}\lambda))$$

where $PP_c^i = \{\lambda_c \mid \langle PP,\lambda_c,\lambda_s\rangle_\vee \in S_i\}$, $PP_s^i = \{\lambda_s \mid \langle PP,\lambda_c,\lambda_s\rangle_\vee \in S_i\}$. If we denote P as the set of all program points in the program, that distance can later be extended to a distance between analyses as $d(S_1,S_2) = \frac{1}{|P|}\sum_{PP\in P} d_{PP}(S_1,S_2)$, or any other combination of the distances $d_{PP}(S_1,S_2)$ (e.g, weighted average, $||\cdot||_2$). This distance is more appropriate for measuring precision than the previous one, but it will still inherit all metric properties except the identity of indiscernibles. An example of this distance can be found in [6].

Tree Distance. For the third distance, we propose the following recursive definition, which can easily be translated into an algorithm:

$$d(T_1, T_2) = \begin{cases} \mu \frac{1}{2}(d_\alpha(\lambda_c^1, \lambda_c^2) + d_\alpha(\lambda_s^1, \lambda_s^2)) + (1-\mu)\frac{1}{|C|}\sum_{(c_1,c_2)\in C} d(c_1, c_2) & if\ C \neq \emptyset \\ else \quad \frac{1}{2}(d_\alpha(\lambda_c^1, \lambda_c^2) + d_\alpha(\lambda_s^1, \lambda_s^2)) \end{cases}$$

where $T_1 = \langle P, \lambda_c^1, \lambda_s^1 \rangle$, $T_2 = \langle P, \lambda_c^2, \lambda_s^2 \rangle$, $\mu \in (0, 1]$ and $C = \{(c_1, c_2) \mid c_1 \in ch(T_1), c_2 \in ch(T_2), val(c_1) = \langle X, \_, \_ \rangle, val(c_2) = \langle Y, \_, \_ \rangle, X = Y\}$.

This definition is possible because the two AND-OR trees, when considering their infinite, non cyclic representation, will necessarily have the same shape, and therefore we are always comparing a node with its correspondent node in the other tree. That shape could only differ for real analysis if one of them detects and unreachable trace (e.g., a clause not applicable or a literal after another that fails), but not the other, thus having one subtree in the second not occurring in the first. But that can also be modelled as that subtree occurring also in the first with every abstract substitution being $\perp$.

This distance is well defined, even if the trees, and therefore the recursions, are infinite, since the expression above always converges. To back up that claim we provide the following argument, which also shows how that distance can be easily computed in finite time. Since the AND-OR trees always have a finite representation as cyclic trees with n and m nodes respectively, there are at most $n * m$ different pairs of nodes to visit during the recursion. Assigning a variable to each pair that is actually visited, the recursive expression can be expressed as a linear system of equations. That system has a unique solution since there is an equation for each variable and the associated matrix, which is therefore squared, has strictly dominant diagonal. An example of this can be found in [6].

The idea of this distance is that we consider more relevant the distance between the upper nodes than the distance between the deeper ones, but we still consider all of them and do not miss any of the analysis information. As a result, this distance will directly inherit the identity of indiscernibles (apart from all other metric properties) from d_α.

5 Experimental Evaluation

To evaluate the usefulness of the program analysis distances, we set up a practical scenario in which we study quantitatively the cost and precision tradeoff for several abstract domains. We propose the following methodology to measure that precision:

Base Domain. Recall that in the distances defined so far, we assume that we compare two analyses using the same abstract domain. We relax this requirement by translating each analysis to a common *base domain*, rich enough to reflect a particular program property of interest. An abstract substitution λ over a domain D_α is translated to a new domain $D_{\alpha'}$ as $\lambda' = \alpha'(\gamma(\lambda))$, and the AND-OR tree is translated by just translating any abstract substitution occurring in it. The results still over-approximates concrete executions, but this time all over the same abstract domain.

Program Analysis Intersection. Ideally we would compare each analysis with the actual semantics of a program for a given abstract query, represented also as an AND-OR tree. However, this semantics is undecidable in general, and we are seeking an automated process. Instead, we approximated it as the *intersection* of all the computed analyses. The intersection between two trees, which can be easily generalized to n trees, is defined as $\mathit{inter}(T_1, T_2) = T$, with

$$\begin{aligned}
\mathit{val}(T_1) = \langle X, \lambda_c^1, \lambda_s^2 \rangle,\ \mathit{val}(T_2) &= \langle X, \lambda_c^2, \lambda_s^2 \rangle,\ \mathit{val}(T) = \langle X, \lambda_c^1 \sqcap \lambda_c^2, \lambda_s^1 \sqcap \lambda_s^2 \rangle \\
\mathit{ch}(T) \quad = \{\ \mathit{inter}(c_1, c_2) \mid c_1 &\in \mathit{ch}(T_1),\quad c_2 \in \mathit{ch}(T_2),\quad \mathit{val}(c_1) = \langle X, \_, \_ \rangle, \\
&\mathit{val}(c_2) = \langle Y, \_, \_ \rangle,\quad X = Y \}
\end{aligned}$$

That is, a new AND-OR tree with the same shape as those computed by the analyses, but where each abstract substitution is the greatest lower bound of the corresponding abstract substitutions in the other trees. The resulting tree is the least general AND-OR tree we can obtain that still over-approximates every concrete execution. We can now use that tree to measure the (loss of) precision of an analysis as its distance to the tree, being that distance 0 is the analysis is as precise as the intersection, and growing up to 1 as it gets more imprecise.

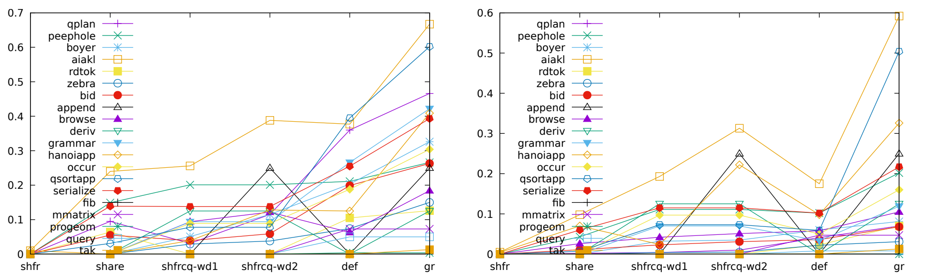

Fig. 2. (a) Precision using flat distance and (b) tree distance (micro-benchmarks)

Case Study: Variable Sharing Domains. We have applied the method above on a well known set of (micro-)benchmarks for CLP analysis, and a number of modules from a real application (the LPdoc documentation generator). The programs are analyzed using the CiaoPP framework [20] and the domains *shfr* [33], *share* [24,32], *def* [15,1], and *sharefree_ clique* [34] with different widenings. All these domains express sharing between variables among other things, and we compare them with respect to the base *share* domain. All experiments are run on a Linux machine with Intel Core i5 CPU and 8GB of RAM.

Fig. 2 and Fig. 3 show the results for the micro-benchmarks. Fig. 4 and Fig. 5 show the same experiment on LPdoc modules. In both experiments we measure the precision using the flat distance, tree distance, and top distance. In general,

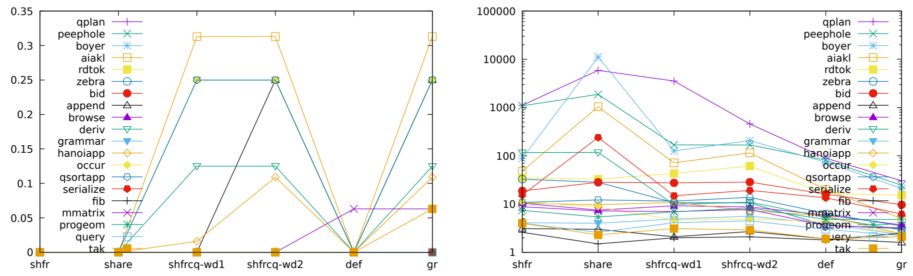

Fig. 3. (a) Precision using top distance and (b) Analysis time (micro-benchmarks)

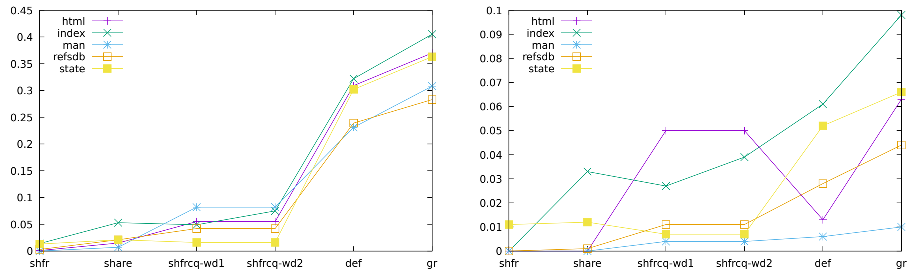

Fig. 4. (a) Precision using flat distance and (b) tree distance (LPdoc benchmark)

the results align with our a priori knowledge: that *shfr* is strictly more precise than all other domains, but can sometimes be slower; while *gr* is less precise and generally faster. As expected, the flat and tree distances show that *share* is in all cases less precise than *shfr*, and not significantly cheaper (sometimes even more costly). The tree distance shows a more pronounced variation of precision when comparing *share* and widenings. While this can also be appreciated in the top distance, the top distance fails to show the difference between *share* and *shfr*. Thus, the tree distance seems to offer a good balance. For small programs where analysis requires less than 100ms in *shfr*, there seems to be no advantage in using less precise domains. Also as expected, for large programs widenings provide significant speedups with moderate precision lose. Small programs do not benefit in general from widenings. Finally, the *def* domain shows very good

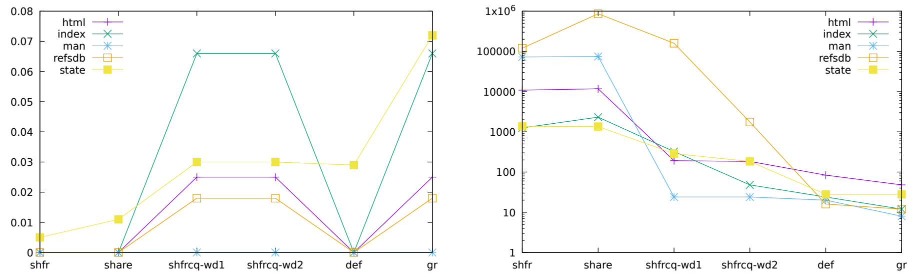

Fig. 5. (a) Precision using top distance and (b) Analysis time (LPdoc benchmarks)

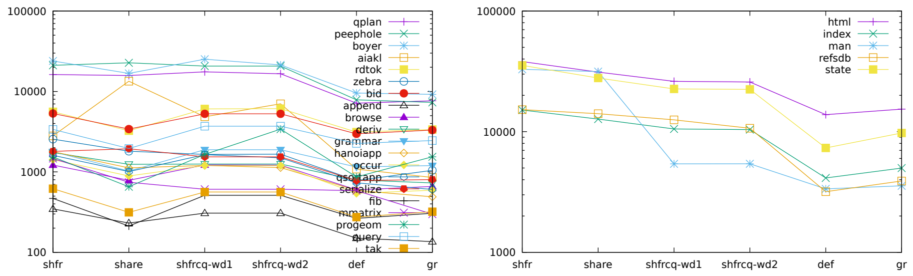

Fig. 6. (a) Analysis size (micro-benchmarks) and (b) Analysis size (LPdoc benchmark)

precision w.r.t. the top distance, representing that the domain is good enough to capture the behavior of predicates at the module interface for the selected benchmarks.

Fig. 6 reflects the size of the AND-OR tree and experimentally it is correlated with the analysis time. The size measures the cost of representing abstract substitutions as Prolog terms (roughly as the number of functor and constant symbols).

6 Related Work

Distances in lattices: Lattices and other structures that arise from order relations are common in many areas of computer science and mathematics, so it is not surprising that there have been already some attempts at proposing metrics in them. E.g., [18] has a dedicated chapter for metrics in lattices, Horn and Tarski [22] studied measures in boolean algebras. *Distances among terms:* Hutch [23], Nienhuys-Cheng [36] and Jan Ramon [38] all propose distances in the space of terms and extend them to distances between sets of terms or clauses. Our proposed distance for *regular types* can be interpreted as the abstraction of the distance proposed by Nienhuys-Cheng. Furthermore, [38] develop some theory of metrics in partial orders, as also does De Raedt [11]. *Distances among abstract elements and operators:* Logozzo [28] proposes defining metrics in partially ordered sets and applying them to quantifying the relative loss of precision induced by numeric abstract domains. Our work is similar in that we also propose a notion of distance in abstract domains. However, they restrict their proposed distances to finite or numeric domains, while we focus instead on logic programming-oriented, possible infinite, domains. Also, our approach to quantifying the precision of abstract interpretations follows quite different ideas. They use their distances to define a notion of error induced by an abstract value, and then a notion of error induced by a finite abstract domain and its abstract operators, with respect to the concrete domain and concrete operators. Instead, we work in the context of given programs, and quantify the difference of precision between the results of different analyses for those programs, by extending our metrics in abstract domains to metrics in the space of abstract executions of a program and comparing those results. Sotin [39] defines measures in $\mathbb{R}^n$ that allow quantifying the difference in precision between two abstract values of a numeric domain, by comparing the size of their concretizations. This is applied to guessing the most appropriate domain to analyse a program, by under-approximating the potentially visited states via random testing and comparing the precision with which different domains would approximate those states. Di Pierro [13] proposes a notion of probabilistic abstract interpretation, which allows measuring the precision of an abstract domain and its operators. In their proposed framework, abstract domains are vector spaces instead of partially ordered sets, and it is not clear whether every domain, and in particular those used in logic programming, can be reinterpreted within that framework. Cortesi [7] proposes a formal methodology to compare qualitatively the precision of two abstract domains with respect to some of the information they express, that is, to know if one is strictly more precise that the other according to only part of the properties they abstract. In our experiments, we compare the precision of different analyses with respect to some of the information they express. For some, we know that one is qualitatively more precise than the other in Cortesi's paper's sense, and that is reflected in our results.

7 Conclusions

We have proposed a new approach for measuring and comparing precision across different analyses, based on defining distances in abstract domains and extending them to distances between whole analyses. We have surveyed and extended previous proposals for distances and metrics in lattices or abstract domains, and proposed metrics for some common (C)LP domains. We have also proposed extensions of those metrics to the space of whole program analysis. We have implemented those metrics and applied them to measuring the precision of different sharing-related (C)LP analyses on both benchmarks and a realistic program. We believe that this application of distances is promising for debugging the precision of analyses and calibrating heuristics for combining different domains in portfolio approaches, without prior knowledge and treating domains as black boxes (except for the translation to the *base* domain). In the future we plan to apply the proposed concepts in other applications beyond measuring precision in analysis, such as studying how programming methodologies or optimizations affect the analyses, comparing obfuscated programs, giving approximate results in semantic code browsing [14], program synthesis, software metrics, etc.

References

1. Armstrong, T., Marriott, K., Schachte, P., Søndergaard, H.: Boolean functions for dependency analysis: Algebraic properties and efficient representation. In: Springer-Verlag (ed.) Static Analysis Symposium, SAS'94

2. Banda, G., Gallagher, J.P.: Analysis of Linear Hybrid Systems in CLP. In: Hanus, M. (ed.) LOPSTR. LNCS, vol. 5438, pp. 55–70. Springer (2009)
3. Bjørner, N., Gurfinkel, A., McMillan, K.L., Rybalchenko, A.: Horn Clause Solvers for Program Verification. In: Essays to Yuri Gurevich on his 75th Birthday. pp. 24–51 (2015)
4. Bruynooghe, M.: A Practical Framework for the Abstract Interpretation of Logic Programs. Journal of Logic Programming **10**, 91–124 (1991)
5. Bueno, F., García de la Banda, M., Hermenegildo, M.V.: Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In: International Symposium on Logic Programming. pp. 320–336. MIT Press (November 1994), ftp://cliplab.org/pub/papers/effofabs.ps.Z
6. Casso, I., Morales, J.F., Lopez-Garcia, P., Hermenegildo, M.V.: Computing Abstract Distances in Logic Programs. Tech. Rep. CLIP-2/2019.0, The CLIP Lab, IMDEA Software Institute and T.U. Madrid (July 2019), http://arxiv.org/abs/1907.13263
7. Cortesi, A., File, G., Winsborough, W.: Comparison of Abstract Interpretations. In: Nineteenth International Colloquium on Automata, Languages, and Programming. vol. 623. Springer-Verlag LNCS (1992)
8. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of POPL'77. pp. 238–252. ACM Press (1977)
9. Dart, P., Zobel, J.: A Regular Type Language for Logic Programs. In: Types in Logic Programming, pp. 157–187. MIT Press (1992)
10. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: VeriMAP: A Tool for Verifying Programs through Transformations. In: TACAS, ETAPS. pp. 568–574 (2014)
11. De Raedt, L., Ramon, J.: Deriving distance metrics from generality relations. Pattern Recogn. Lett. **30**(3), 187–191 (Feb 2009). https://doi.org/10.1016/j.patrec.2008.09.007, http://dx.doi.org/10.1016/j.patrec.2008.09.007
12. Debray, S.K.: Static Inference of Modes and Data Dependencies in Logic Programs. ACM Transactions on Programming Languages and Systems **11**(3), 418–450 (1989)
13. Di Pierro, A., Wiklicky, H.: Measuring the precision of abstract interpretations. In: Logic Based Program Synthesis and Transformation. pp. 147–164. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
14. Garcia-Contreras, I., Morales, J.F., Hermenegildo, M.V.: Semantic Code Browsing. TPLP (ICLP'16 Special Issue) **16**(5-6), 721–737 (October 2016)
15. García de la Banda, M., Hermenegildo, M.V.: A Practical Application of Sharing and Freeness Inference. In: 1992 Workshop on Static Analysis WSA'92. pp. 118–125. No. 81–82 in BIGRE, IRISA-Beaulieu, Bourdeaux, France (September 1992)
16. García de la Banda, M., Hermenegildo, M.V., Bruynooghe, M., Dumortier, V., Janssens, G., Simoens, W.: Global Analysis of Constraint Logic Programs. ACM TOPLAS **18**(5) (1996)
17. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A Software Verifier Based on Horn Clauses. In: TACAS. pp. 549–551 (2012)
18. Grätzer, G.: General Lattice Theory, second edition (01 1998). https://doi.org/10.1007/978-3-0348-7633-9
19. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: CAV. pp. 343–361 (2015)

20. Hermenegildo, M., Puebla, G., Bueno, F., Lopez-Garcia, P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Comp. Progr. **58**(1–2) (2005)
21. Hermenegildo, M., Warren, R., Debray, S.K.: Global Flow Analysis as a Practical Compilation Tool. JLP **13**(4), 349–367 (August 1992)
22. Horn, A., Tarski, A.: Measures in boolean algebras. Transactions of the American Mathematical Society **64**(3), 467–497 (1948)
23. Hutchinson, A.: Metrics on terms and clauses. In: van Someren, M., Widmer, G. (eds.) Machine Learning: ECML-97. pp. 138–145. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
24. Jacobs, D., Langen, A.: Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In: North American Conference on Logic Programming (1989)
25. Kafle, B., Gallagher, J.P., Morales, J.F.: RAHFT: A Tool for Verifying Horn Clauses Using Abstract Interpretation and Finite Tree Automata. In: CAV. pp. 261–268 (2016)
26. Kelly, A., Macdonald, A., Marriott, K., Sondergaard, H., Stuckey, P., Yap, R.: An optimizing compiler for CLP(R). In: Proc. of Constraint Programming (CP'95). LNCS, Springer (1995)
27. Liqat, U., Kerrison, S., Serrano, A., Georgiou, K., Lopez-Garcia, P., Grech, N., Hermenegildo, M.V., Eder, K.: Energy Consumption Analysis of Programs based on XMOS ISA-level Models. In: LOPSTR. LNCS, vol. 8901, pp. 72–90. Springer (2014)
28. Logozzo, F., Popeea, C., Laviron, V.: Towards a quantitative estimation of abstract interpretations (extended abstract). In: Workshop on Quantitative Analysis of Software (June 2009)
29. Madsen, M., Yee, M., Lhoták, O.: From Datalog to FLIX: a Declarative Language for Fixed Points on Lattices. In: PLDI, ACM. pp. 194–208 (2016)
30. Marriott, K., Søndergaard, H., Jones, N.: Denotational Abstract Interpretation of Logic Programs. ACM Transactions on Programming Languages and Systems **16**(3), 607–648 (1994)
31. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: LOPSTR. LNCS, vol. 4915, pp. 154–168. Springer-Verlag (August 2007)
32. Muthukumar, K., Hermenegildo, M.: Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In: NACLP'89. pp. 166–189. MIT Press (October 1989)
33. Muthukumar, K., Hermenegildo, M.: Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In: ICLP'91. pp. 49–63. MIT Press (June 1991)
34. Navas, J., Bueno, F., Hermenegildo, M.V.: Efficient Top-Down Set-Sharing Analysis Using Cliques. In: 8th Int'l. Symp. on Practical Aspects of Declarative Languages (PADL'06). pp. 183–198. No. 2819 in LNCS, Springer (January 2006)
35. Navas, J., Méndez-Lojo, M., Hermenegildo, M.V.: User-Definable Resource Usage Bounds Analysis for Java Bytecode. In: BYTECODE'09. ENTCS, vol. 253. Elsevier (March 2009)
36. Nienhuys-Cheng, S.H.: Distance between herbrand interpretations: A measure for approximations to a target concept. In: Lavrač, N., Džeroski, S. (eds.) Inductive Logic Programming. pp. 213–226. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)

37. Puebla, G., Bueno, F., Hermenegildo, M.V.: An Assertion Language for Constraint Logic Programs. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) Analysis and Visualization Tools for Constraint Programming, pp. 23–61. No. 1870 in LNCS, Springer-Verlag (September 2000)
38. Ramon, J., Bruynooghe, M.: A framework for defining distances between first-order logic objects. In: 8th Intl. WS on Inductive Logic Programming. pp. 271–280 (1998)
39. Sotin, P.: Quantifying the Precision of Numerical Abstract Domains. Research report, INRIA (Feb 2010), https://hal.inria.fr/inria-00457324
40. Van Roy, P., Despain, A.: High-Performance Logic Programming with the Aquarius Prolog Compiler. IEEE Computer Magazine pp. 54–68 (January 1992)